MDP Toolbox car race example – Sparse matrices illustration

Pierre-Matthieu Pair, Régis Sabbadin

3 février 2005

Résumé

This student report describes the car race example implementation in the MDP Toolbox, as well as the sparse matrices handling in the MDPtoolbox v2.0.

Table des matières

Introduction

L'étude de la planification dans l'incertain plus particulièrement basée sur la *Théorie de la Décision* a connu un essor important en *Intelligence Artificielle* ces dix dernières années. L'apport principal de la Théorie de la Décision au paradigme de la planification dite "classique" en IA est lié à sa faculté de modéliser explicitement l'incertitude et les préférences sur les effets des actions, via des distributions de probabilité et des fonctions d'utilité par exemple. Ceci permet une recherche de plans (ou règles de décisions) plus souple que dans la planification classique : on recherche un plan *maximisant* un certain critère, plutôt qu'un plan permettant *à coup sûr* d'atteindre tel ou tel objectif fixé. Typiquement, le critère à optimiser sera l'espérance mathématique d'une variable aléatoire représentant une somme de récompenses perçues au cours du temps dans un problème de décision séquentielle. Nous verrons que d'autres critères peuvent également être utilisés.

La plupart des travaux récents en IA sur la planification basée sur la Théorie de la Décision utilisent le cadre des *Processus Décisionnels Markoviens* (PDM), initialement étudié dans la communauté *Recherche Opérationnelle*. Ces travaux ont appliqué, adapté ou étendu le cadre des PDM pour en faire bénéficier différents champs de la Planification en IA : Planification dans l'incertain en environnement complètement ou partiellement observable, apprentissage de plans, représentation structurée des problèmes de planification et des plans, etc.

L'unité BIA de l'INRA Toulouse a développé une boîte à outils Matlab, la MDP Toolbox, contenant un certain nombre d'éléments permettant la représentation et la résolution de PDM. Le propos de mon stage a été de me familiariser avec les PDM et les différents algorithmes de résolution associés. Dans un premier temps j'ai implémenté le problème de course automobile, un PDM de grande taille, modifié la MDP Toolbox pour lui permettre de résoudre des systèmes creux de grande taille, et résolu le problème en utilisant les algorithmes existants dans la Toolbox. Je me suis ensuite familiarisé avec deux algorithmes de résolution avancés, le TD(λ), et le Q-learning, que j'ai implémentés en vue de leur intégration dans la nouvelle version de la MDP Toolbox.

Chapitre 1

Les Processus Décisionnels de Markov

1.1 Formulation du problème

Dans sa formulation classique, un Processus Décisionnels de Markov est décrit par un quadruplet $< S, A, p, r >$, où S représente l'ensemble des états possibles du système, A l'ensemble des actions applicables, p la probabilité de transition entre états et r une fonction de récompense "immédiate".

1.1.1 États, actions, transitions et politiques

Dans le cas de problèmes à ensemble d'états fini, auquel nous nous limitons, $S = \{s_1, \ldots, s_n\}$ est un ensemble fini représentant les états que peut parcourir le système au cours du temps. Il peut arriver que le système observé ne soit que *partiellement observable*, auquel cas l'état du système n'est pas connu précisément à chaque instant, mais décrit par une distribution de probabilité b (pour *belief state*, *état de croyance*, en français) sur S. $A = \{a_1, \ldots, a_m\}$ représente l'ensemble des actions applicables à chaque instant, dont l'application va modifier l'état du système.

Un processus stochastique à temps discret est constitué d'un espace d'états S et d'un ensemble d'actions A, mais aussi de distributions de probabilité régissant les transitions entre états. Dans un PDM, l'état courant du processus évolue au cours du temps, l'état s_{t+1} à l'étape $t+1$ dépendant (stochastiquement) de l'état s_t à l'instant t et de l'action a_t appliquée à cet instant. La probabilité de transition de s_t à s_{t+1} est $p_t(s_{t+1}|s_t, a_t)$. Un tel

processus stochastique est dit *Markovien*, quand la probabilité $p_t(s_{t+1}|s_t, a_t)$ est indépendante de la trajectoire $< s_0, a_0, s_1, \ldots, s_{t-1}, a_{t-1} >$ suivie auparavant. Les probabilités de transition peuvent dépendre du paramètre t ou non. Quand elles ne dépendent pas du paramètre t, le processus est dit *stationnaire*. L'ensemble $H \subseteq \mathrm{IN}$ des étapes du processus est appelé horizon, et peut être fini ou infini. Quand l'horizon est infini, le processus est en général stationnaire.

Nous verrons dans le paragraphe suivant que résoudre un PDM revient à trouver une action "optimale" pour chaque état possible du système, au sens d'un certain critère. Une fonction $\delta : S \times H \rightarrow A$ associant à chaque instant une action à chaque état est appelée *règle de décision* ou *politique*. Une politique est *stationnaire* lorsqu'elle ne dépend pas de l'instant t courant.

1.1.2 Récompense, critère, fonction de valeur, politique optimale

Dans un PDM, en plus du modèle de transition déjà décrit, on définit pour chaque étape t une fonction *récompense* $r_t : S \times A \times S \rightarrow \mathrm{IR}$. $r_t(s, a, s')$ est la récompense perçue lorsque l'action a a été effectuée dans l'état s à l'instant t et a débouché sur l'état s'.

La notion de récompense peut être généralisée sur une trajectoire τ :
$\tau =< s_0, a_0, s_1, \ldots, s_t, a_t, \ldots >, t \in H : V(\tau) = \sum_{t \in H} r_t(s_t, a_t, s_{t+1})$.

Lorsque H est infini la somme précédente ne converge pas forcément et il est courant d'utiliser plutôt la somme γ-pondérée :
$V(\tau) = \sum_{t \in H} \gamma^t r_t(s_t, a_t, s_{t+1})$ qui converge à coup sûr si $0 \leq \gamma < 1$.

Appliqué à la chaîne de Markov associée à une politique δ dans un PDM, ce critère devient :

$$V_\delta(s_0) = E[\sum_{t \in H} \gamma^t r_t(s_t, \delta(s_t), s_{t+1})] \tag{1.1}$$

la moyenne étant prise sur toutes les trajectoires τ possibles en appliquant δ en partant de l'état initial s_0.

Le problème de la recherche d'une politique optimale peut s'écrire :

$$\text{Trouver } \delta^*, S \rightarrow A, V_{\delta^*}(s) \geq V_\delta(s), \forall s \in S, \forall \delta \in A^S \tag{1.2}$$

Ce problème est classiquement résolu par des méthodes de type *Programmation Dynamique Stochastique*. Les algorithmes de *Recherche Arrière*,

Itération de la Politique et *Itération de la Valeur* sont les plus couramment utilisés. Nous les décrivons dans la section suivante.

1.2 Algorithmes classiques de résolution des PDM

1.2.1 Horizon fini : Recherche arrière par induction

Dans le cas où $H = \{0, \ldots, N\}$ est fini, on peut calculer $V_{\delta,t}$ la valeur de la politique δ en tout état s à l'instant t grâce au système d'équations suivant :

$$V_{\delta,t}(s) = \sum_{s' \in S} p_t(s'|s, \delta(s)) \cdot (r_t(s, \delta(s), s') + \gamma \cdot V_{\delta,t+1}(s')), t < N, s \in S,$$

$$V_{\delta,N}(s) = 0, \forall s \in S \quad (1.3)$$

$V_{\delta,t}(s)$ peut ainsi être calculé pour tout s, t itérativement en partant de la fin.

Une politique optimale δ^* peut être également calculée itérativement, comme suit :

$$\delta^*(s,t) = argmax_{a \in A} \sum_{s' \in S} p_t(s'|s,a) \cdot (r_t(s,a,s') + \gamma \cdot V^*_{t+1}(s'))$$

$$V^*_t(s) = \max_{a \in A} \sum_{s' \in S} p_t(s'|s,a) \cdot (r_t(s,a,s') + \gamma \cdot V^*_{t+1}(s'))$$

$$V^*_N(s) = 0, \forall s. \quad (1.4)$$

1.2.2 Horizon infini : Itération de la Valeur et Itération de la politique

Lorsqu'on passe à un horizon infini (pour des problèmes stationnaires), on peut montrer que $V_\delta(s)$, valeur de la politique δ en s est indépendante du temps et vérifie le système d'équations:

$$V_\delta(s) = \sum_{s' \in S} p(s'|s, \delta(s)) \cdot (r(s, \delta(s), s') + \gamma \cdot V_\delta(s')), \forall s. \quad (1.5)$$

Ce système peut être résolu par une méthode de type simplexe, ou par un algorithme itératif de type *approximation successive*:

$$V^0_\delta(s) = 0 \forall s,$$

$$V^n_\delta(s) = \sum_{s' \in S} p(s'|s, \delta(s)) \cdot (r(s, \delta(s), s') + \gamma \cdot V^{n-1}_\delta(s')) \forall s. \quad (1.6)$$

Lorsque $n \to \infty$, $V_\delta^n \to V_\delta$ avec vitesse de convergence et erreur bornée en n.

Itération de la valeur

L'algorithme *Itération de la valeur* s'inspire très fortement du schéma itératif 1.6 pour calculer une politique optimale :

$$V_\delta^0(s) = 0 \forall s,$$
$$V_\delta^n(s) = \max_{a \in A} \sum_{s' \in S} p(s'|s,a) \cdot (r(s,a,s') + \gamma \cdot V_\delta^{n-1}(s')), \forall s. \quad (1.7)$$

Il existe un n à partir duquel les actions maximisant la partie droite de l'équation 1.7 forment une politique optimale. On recense un ensemble de critères d'arrêt pour l'algorithme d'itération de la valeur. Notons que

$$\delta(s) = argmax_{a \in A} \sum_{s' \in S} p(s'|s,a) \cdot (r(s,\delta(s,a,s') + \gamma \cdot V^*(s')) \forall s. \quad (1.8)$$

Itération de la politique

L'algorithme *Itération de la politique* s'inspire également du schéma 1.6 pour calculer une politique optimale. Plus précisément, il consiste en l'alternance de phases *d'évaluation* et *d'amélioration* de la politique courante. La phase *d'évaluation* évalue la politique courante δ via l'un des algorithmes 1.5 ou 1.6. La phase *d'amélioration* transforme la politique δ en une politique δ' "meilleure" ($V_{\delta'}(s) \geq V_\delta(s), \forall s$) :

$$\delta'(s) = \max_{a \in A} \sum_{s' \in S} p(s'|s,a) \cdot (r(s,a,s') + \gamma \cdot V_\delta(s')), \forall s. \quad (1.9)$$

Le critère d'arrêt de l'algorithme *Itération de la politique* est l'égalité des politiques δ et δ'. Cet algorithme converge en général en un très petit nombre d'itérations (appels à la phase d'amélioration), mais chaque itération est coûteuse (résolution d'un système d'équations).

L'algorithme *Itération de la politique modifiée* améliore l'algorithme *Itération de la politique* en utilisant le schéma d'approximation successive 1.6, mais en opérant un nombre limité d'étapes (n petit). En pratique, l'algorithme *Itération de la politique modifiée*, avec un bon choix du nombre d'itérations de la phase d'évaluation, est bien plus efficace que les algorithmes *Itération de la politique* et *Itération de la valeur*. Notons également qu'en changeant le paramètre "nombre d'étapes", on peut retrouver ces deux algorithmes : Lorsque $n = 1$ on retrouve l'algorithme d'itération de la valeur et lorsque n tend vers l'infini, on retrouve l'algorithme d'itération de la politique.

1.3 Exemple : Course de voiture

Pour illustrer le cadre des Processus Décisionnels Markoviens, nous utiliserons un exemple. Il s'agit d'un jeu de simulation de course de voiture, qui se joue habituellement sur une feuille de papier quadrillée sur laquelle est dessiné un circuit. La figure 5.1 représente un tel circuit (la ligne de départ se trouve en bas à gauche, la ligne d'arrivée à droite).

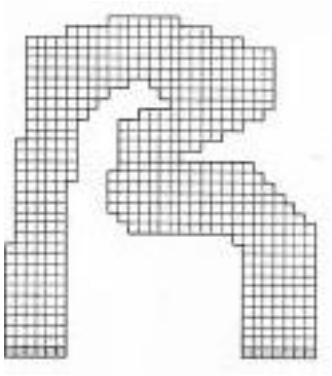


Figure 1.1: Le circuit de l'exemple de course de voitures.

A chaque instant t, l'état s_t de la voiture est représenté par sa position (coordonnées : (x_t, y_t)) et sa vitesse (coordonnées $(\dot{x}_t, \dot{y}_t)$). Les actions disponibles consistent à modifier le vecteur vitesse (par des coups de volant, de frein ou d'accélérateur...). Plus précisément, à chaque instant il est possible de choisir un vecteur accélération $(ax_t, ay_t) \in \{-1, 0, 1\} \times \{-1, 0, 1\}$. Pour introduire un peu d'aléa, il a été proposé qu'avec une certaine probabilité p, la commande ne soit pas transmise à la voiture (à cause d'un problème d'aquaplanning de la voiture par exemple). Donc, avec une probabilité $1-p$, la commande (ax_t, ay_t) est bien transmise, et avec probabilité p, c'est la commande $(0, 0)$ qui est transmise. Les équations suivantes gouvernent le mouvement de la voiture :

$$\begin{aligned} x_{t+1} &= x_t + \dot{x}_t + ax_t, \\ y_{t+1} &= y_t + \dot{y}_t + ay_t, \\ \dot{x}_{t+1} &= \dot{x}_t + ax_t, \\ \dot{y}_{t+1} &= \dot{y}_t + ay_t. \end{aligned} \tag{1.10}$$

Le vecteur d'état de la voiture comporte une cinquième variable, n_t, représentant un "nombre d'accidents", et servant à modéliser les sorties de circuit

éventuelles de la voiture. En effet, si à un instant donné le vecteur d'origine (x_t, y_t) et de coordonnées $(\dot{x}_t, \dot{y}_t)$ coupe la frontière du circuit, n_t est incrémenté et la voiture est immobilisée :

$$\begin{aligned} x_{t+1} &= x_t, \\ y_{t+1} &= y_t, \\ \dot{x}_{t+1} &= 0, \\ \dot{y}_{t+1} &= 0, \\ n_{t+1} &= n_t + 1. \end{aligned}$$

Lorsque la voiture à franchi la ligne d'arrivée ou si le nombre d'accidents dépasse une limite fixée N_{acc}, la voiture atteint un état absorbant[1].

L'objectif consiste bien entendu à rallier la ligne d'arrivée le plus rapidement possible en partant d'une position quelconque sur la ligne de départ. Ainsi, outre la dynamique du système, le modèle du problème de course de voiture comporte un modèle de récompenses, servant à modéliser cet objectif. La récompense $r_t(s_t, a_t, s_{t+1})$ est définie par :

$$\begin{aligned} r_t(s_t, a_t, s_{t+1}) &= \Delta T \text{ si la trajectoire ne coupe ni le circuit ni la ligne d'arrivée,} \\ r_t(s_t, a_t, s_{t+1}) &= T_{acc} \text{ si la trajectoire coupe le circuit,} \\ r_t(s_t, a_t, s_{t+1}) &= T_\infty \text{ si la trajectoire coupe le circuit pour la } N_{acc}\text{ème fois,} \\ r_t(s_t, a_t, s_{t+1}) &= \alpha \cdot \Delta T \text{ si la trajectoire coupe la ligne d'arrivée.} \end{aligned}$$

α est la proportion du vecteur déplacement qui se trouve avant la coupure de la ligne d'arrivée.

Le critère *à minimiser* (et non à maximiser) est le critère total $\sum_{t=1..\infty} r_t(s_t, a_t, s_{t+1})$, représentant le temps utilisé pour franchir la ligne d'arrivée. Notons qu'en général les algorithmes classiques ne convergent pas forcément lorsque le critère total est utilisé en horizon infini. Néanmoins, lorsqu'il existe une politique de valeur finie en tous points les algorithmes d'itération de la valeur ou de la politique la trouvent.

Le problème de course de voiture peut être décrit dans le cadre PDM que nous venons de présenter. Le PDM obtenu comporte plus de vingt mille états lorsque N_{acc} vaut 1. Les limites de la Programmation Dynamique classique sont dès lors atteintes puisque les algorithmes classiques ne permettent pas

[1]Un état absorbant dans un PDM est tout simplement un état dans lequel on a une probabilité 1 de rester, quelle que soit l'action appliquée, et dont les transitions vers lui-même sont associées à une récompense nulle.

de trouver une politique optimale en un temps raisonable au dela de quelques dizaines de milliers d'états.

Après un détour via la description de la notion d'observabilité partielle dans les PDMs, nous décrirons deux types d'approches permettant de résoudre ce type de PDM "de grande taille" : *la Programmation Dynamique temps-réel* et *les représentations structurées.*

1.4 Extensions des Processus Décisionnels Markoviens proposées en Intelligence Artificielle

Le cadre des Processus Décisionnels Markoviens s'est assez largement imposé comme modèle pour la planification dans l'incertain en Intelligence Artificielle ces dernières années. Néanmoins, un certain nombre de limitations inhérentes au modèle le rendent insuffisant pour modéliser et résoudre la plupart des problèmes de planification dans l'incertain. Parmi ces limitations, citons :

- L'hypothèse d'observabilité complète de l'état du monde à chaque instant.
- L'hypothèse (différente de la précédente) de connaissance parfaite du modèle (transitions, récompenses). En effet, parfois ce modèle n'est accessible qu'indirectement, par simulation ou expérimentation. Parfois également, seules des évaluations "qualitatives" des préférences et des connaissances sont disponibles.
- L'hypothèse de représentation des états et décisions *en extension*, beaucoup plus limitée que les langages de représentation habituellement utilisés en planification qui permettent de modéliser des problèmes beaucoup plus complexes. Plus généralement, les PDMs traitent difficilement les problèmes de très grande taille (au delà de quelques millions d'états).

Chapitre 2

Exemple de la course automobile Modélisation des données

2.1 Introduction

L'exemple de la course automobile se prête assez bien à la modélisation puisqu'on peut facilement le considérer comme un problème discret en deux dimensions. Ce chapitre décrit les décisions de modélisation qui ont été prises.

2.2 Représentation du circuit

Le circuit est représenté sous Matlab par une matrice M×N dont les éléments sont:
- 0 si il s'agit de hors-piste
- 1 s'il s'agit de la piste
- 2 s'il s'agit de la ligne de départ/arrivée

A terme, nous pourrons envisager d'introduire de nouveaux types de route où le probabilité de non-transmission de l'ordre pourra varier, ou, de manière plus générale, la route aura des propriétés particulières.

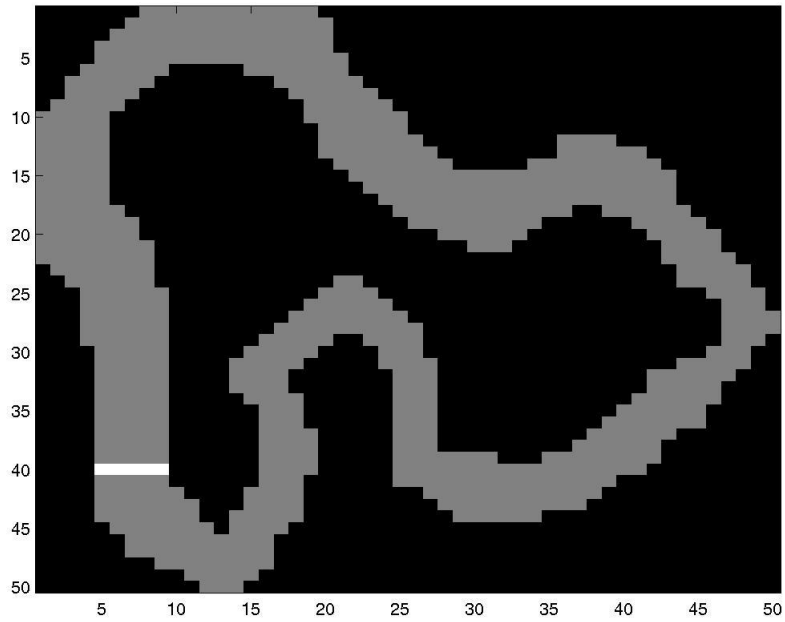


Figure 2.1: Un circuit 50x50.

2.3 Saisie des données

Les circuits étant représentés par des matrices, pour des tailles importantes (plus de 10×10=100 éléments), des problèmes pratiques de saisie apparaissent. Pour y pallier, on a procédé de la manière suivante:
- Création d'une image bitmap noir et blanc où le fond est blanc et le circuit noir
- Enregistrement de cette image au format .txt
- Remplacement des caractères ascii représentant le noir ou blanc par 1 ou 0
- Import sous Matlab

Plusieurs tailles de circuit ont été prévues: 10×10, 25×25, 50×50 et 100×100 cases.

2.4 Modélisation d'un état

L'état d'un système est un concept fondamental dans les PDM, il faut donc bien garder en tête ce dont on a besoin. Dans l'exemple de course automobile, on cherche à optimiser des décisions dans le but de faire un tour de circuit en un temps minimal. Par conséquent, l'information qui nous intéresse est l'état du véhicule par rapport au circuit. L'état du système à un instant donné peut donc être résumé par l'état du véhicule par rapport au circuit et représenté par les quatre variables suivantes:
-X la position selon l'axe des abscisses;
-Y la position selon l'axe des ordonnées;
-VX la vitesse en X;
-VY la vitesse en Y.

2.5 Modélisation d'une action

Les actions possibles à un instant donné sont: accélérer/ralentir, ou donner un coup de volant. Ces actions peuvent être modélisées par l'évolution de (AX,AY) le vecteur accélération de la voiture. On considère que l'on peut faire évoluer chaque composante d'une unité à chaque instant. Les 9 actions possibles sont donc toutes les combinaisons des actions AX=-1,0 ou +1 et AY=-1,0 ou +1.

2.6 Modélisation du problème par un PDM

La formulation d'un problème de PDM usuel revient à la donnée de P une matrice de transition avec P:S×S×A où S est l'ensemble des états et A l'ensemble des actions, ainsi que d'une matrice C:S×S×A des coûts de transition d'un état à l'autre qu'on cherchera à minimiser. Dans la formulation usuelle des PDM, C est une fonction de récompense qu'on cherche à maximiser. Il nous faut donc arriver à résumer notre information d'état (X,Y,VX,VY) en une seule variable d'état et notre action (AX,AY) en une seule variable d'action.

2.6.1 Le cas de l'accélération

Dans un premier temps, une manière évidente de résumer AX et AY en une seule variable est d'ordonner notre ensemble d'actions et de définir une fonction de AX×AY → A. L'ordre choisi est le suivant: (-1,-1)→1; (-1,0)→2; (-1,1)→3; (0,-1)→4; ... ; (1,1)→9.

2.6.2 Le cas de l'état

Nous allons maintenant nous intéresser au problème de résumer (X Y VX VY) en une seule variable.

Nous allons séparer cette simplification en 2 temps: puisque, dans cette première modélisation, toutes les variables sont indépendantes, nous allons considérer séparément les variables d'espace et de vitesse.

2.6.3 Résumé de la position

Les variables X et Y représentent la position de la voiture sur la piste. La piste elle-même est représentée, on l'a vu, par une matrice M. Par conséquent, les variables X et Y reviennent aux indices de colonne et ligne de cette matrice. Pour chaque couple (X,Y), 3 cas se présentent: 'ligne de départ', 'hors-piste' et 'sur la piste'. Il existe également un état 'course terminée', mais aucun couple (X,Y) ne lui correspond. On peut amalgamer les états 'ligne de départ' et 'sur la piste'. Il reste donc 2 possibilités, ce qui revient à dire que les états 'piste' et 'ligne de départ' sont des couples (X,Y) autorisés, et que les états 'hors-piste' sont des couples interdits. L'état 'course terminée' n'est pas représentable sur la matrice. La matrice du circuit revient donc à une matrice booléenne M_Bool. Nous allons pouvoir résumer X et Y en une seule variable grâce à cette propriété.

Nous allons écrire la matrice M_Bool sous forme d'un vecteur V en concaténant verticalement les colonnes de la matrice, puis nous allons considérer le vecteur Position_Index des indices des éléments non nuls du vecteur V. Nous avons donc ramené les deux variables X et Y à une variable d'index de position. La transformation est inversible du moment qu'on connaît les dimensions de la matrice M_Bool.

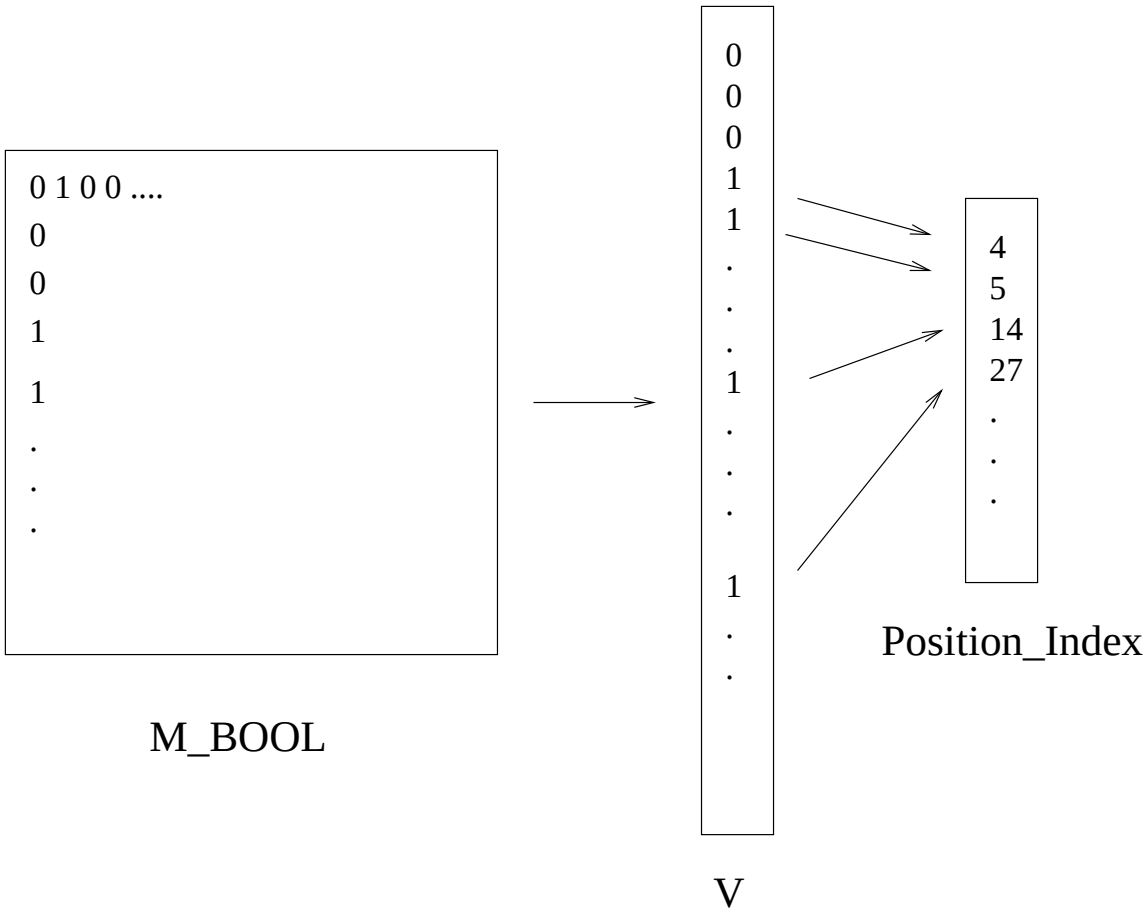


Figure 2.2: La réécriture en une variable.

2.6.4 Résumé de la vitesse

Pour résumer VX et VY, nous suivons un processus similaire: nous considérons que la voiture a une vitesse maximale Vmax. Nous prenons donc une matrice M_Vit de dimensions {-Vmax:Vmax}×{-Vmax:Vmax} dont les éléments sont 1 si la relation

$$VX^2 + VY^2 \leq Vmax^2$$

est vérifiée, et O sinon. A partir de VX, VY et une matrice booléenne, nous pouvons suivre un raisonnement identique à celui suivi pour la position et

arriver à une variable index de vitesse. De même, puisque les dimensions de M_Vit sont connues, la transformation est inversible.

Nous voilà donc avec deux variables d'indices qu'il nous faut résumer en une seule. Pour ce faire, nous allons poser

$$etat = (position\_index) * (nombre\_d\_index\_vitesse) + vitesse\_index$$

Ceci est également inversible dans la mesure où l'on connaît le nombre total d'index de vitesses.

2.6.5 Conclusion: résumé de l'état

En résumé, nous avons ordonné notre espace d'états par rapport aux variables à résumer: pour chaque état de position admissible on considère chaque état de vitesse admissible; la variable d'état est donc l'index, dans cette liste ordonnée, correspondant à la combinaison position/vitesse désirée. On peut également, à partir d'un état, retrouver la position et la vitesse correspondants.

Maintenant que l'état et l'action sont représentés par une variable chacun, nous pouvons appliquer la formulation usuelle pour les PDM. Il va donc falloir calculer la matrice de transition P:S×S×A et la matrice de coût C:S×S. On verra plus loin pourquoi C ne dépend pas de A.

2.7 Calcul des matrices caractéristiques du problème: P la matrice de transition

La matrice de transition P:S×S×A usuelle des PDM est la matrice dont l'élément (i,j,a) correspond à P(j|i,a), autrement dit la probabilité de passer de l'état i à l'état j en ayant choisi l'action a. Il faut donc que i et j balaient l'ensemble des états possibles. Nous avons vu au-dessus comment résumer les états autorisés en une variable d'état, mais il faut penser à rajouter les deux autres états possibles: hors-piste et fin de la course.

L'état 'hors-piste' est considéré comme une unique valeur: en effet, une fois la voiture sortie de la piste, peu importe où, car elle sera soit ramenée à la ligne de départ avec une pénalité, soit envoyée à l'état 'course terminée', selon le choix de modélisation.

L'état 'course terminée' est un état absorbant indiquant le bout de la chaîne.

2.7.1 Calcul des successeurs

Le calcul des successeurs d'un état se fait comme suit: tout d'abord on suppose que l'action a eu l'effet désiré, autrement dit que l'évolution des vecteurs d'accélération s'est faite comme prévu. On calcule alors les nouvelles valeurs de (X Y VX VY) et on vérifie à quel état cela correspond: un état autorisé sur la piste, un état 'hors-piste' ou un état 'fin de la course'. Dans le cas où l'on est dans l'état 'course terminée', le successeur est également 'course terminée' avec une probabilité de 1.

Une fois déterminé le successeur j de (i,a) hors cas particuliers, l'élément correspondant de P est mis à la valeur 1-p, la probabilité que l'action ait eu lieu sans problème. Dans la première modélisation, p est constant pour n'importe quel état. Dans un deuxième temps, il pourra varier selon des paramètres à définir.

Une fois cela fait, on calcule le successeur de l'état en supposant que l'action n'ait pas eu l'effet désiré: cela correspond à l'action (0,0), autrement dit 'commande ignorée' dans la première modélisation. Le calcul de successeur se déroule de manière identique et l'élément correspondant de P est mis à la valeur p, probabilité d'avoir un problème. Dans le cas particulier où l'on se trouve dans l'état 'hors-piste', le successeur est soit un état (X,Y,0,0) où (X,Y) est une case de la ligne de départ déterminée aléatoirement, soit l'état 'hors-piste'.

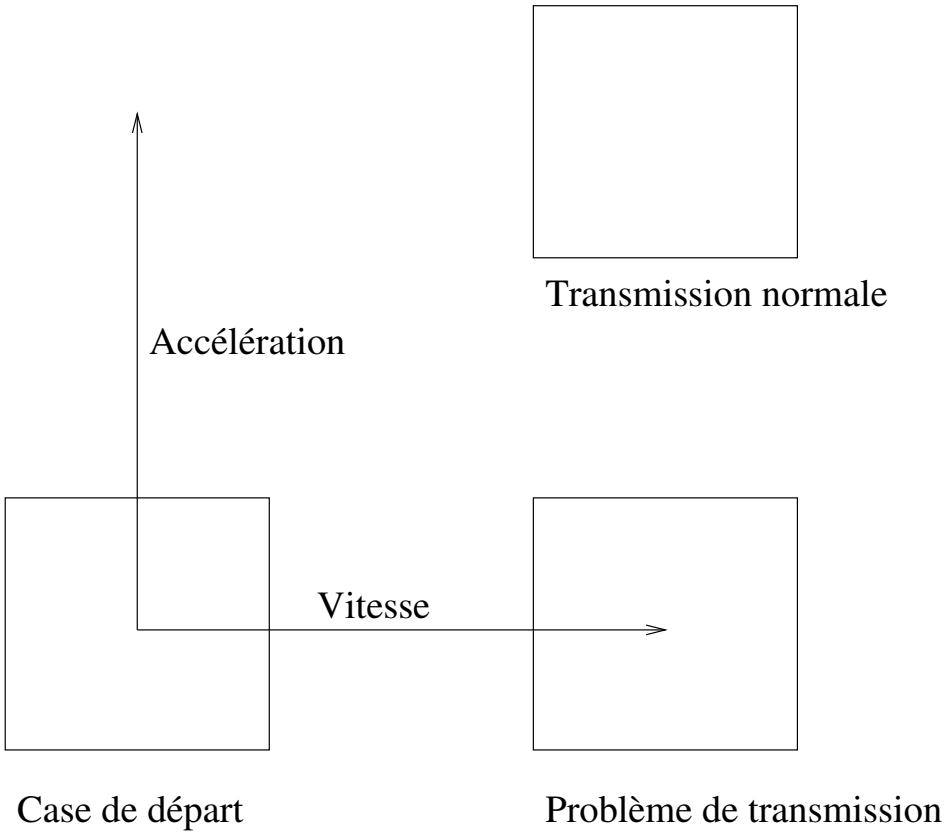


Figure 2.3: Les successeurs d'un état.

2.7.2 Une propriété de P

Il apparaît de ce qui précède que, exceptés les cas 'sortie de piste' et 'course terminée', chaque ligne de P ne contient que 2 valeurs non nulles: le successeur prévu avec une probabilité 1-p et le successeur en cas de problème avec une probabilité p. Les cas particuliers 'hors-piste' et 'course terminée' ont un unique successeur avec une probabilité 1. Ceci est important dans la mesure où la matrice P peut prendre des dimensions extrêmement importantes pour peu que la matrice M soit grande (plus de 10×10). En effet, le fait que P soit très creuse permettra, lors de la phase de codage, d'utiliser un format de matrice creuse, ce qui fera gagner énormément de mémoire.

2.8 Calcul des matrices caractéristiques du problème: C la matrice de coûts

Le calcul de la matrice de coûts est simple: en effet, une transition d'un état autorisé à un autre état (y compris les cas particuliers) représente le passage d'une unité de temps (coût 1).Le passage de l'état 'accident' à la ligne de départ ou à l'état 'hors-piste' comporte une pénalité arbitraire (coût 100).L'état 'course terminée' étant absorbant, il sera toujours son propre successeur avec un coût 0. En réalité, dans notre exemple, le coût de transition ne dépend que de l'état de départ, et pas du successeur. On peut donc résumer la matrice de coûts avec un vecteur V_C=[1,1,...,1,100,0].

Chapitre 3

Codage Matlab, version 1.1

3.1 Introduction

Lors de mon arrivée, il existait déjà la version 1.0 de la 'MDP Toolbox' pour Matlab. Le problème de course automobile utilisant cette toolbox, il m'a semblé raisonnable de commencer ma numérotation à 1.1; de plus, il est rapidement apparu que le code de la toolbox devrait être mis à jour, notamment pour pouvoir supporter le format de matrices creuses, non implémenté jusque-là. La version 1.1 du projet est donc le premier jet du codage du problème, avec une approche "naïve" où le souci principal n'a pas été l'optimisation mais plutôt le bon fonctionnement du code. Les versions suivantes raffineront le codage.

3.2 Liste des fonctions utilisées

Voici une liste des fonctions utilisées, ainsi qu'un bref descriptif de leur utilité. Nous entrerons dans les détails plus loin.

Les variables globales utilisées sont:
VMAX: la vitesse maximale autorisée;
Map_Data: la matrice correspondant au circuit;
Pos_Vector_Indexes: le vecteur des indices de position;
Speed_Vector_Indexes: le vecteur des indices de vitesse;
Finish_Data: le vecteur contenant les informations sur la ligne d'arrivée et
Policy: la politique déterminée par la résolution du PDM.

[P,C,cpu_time]=
gen_transition_matrix(file_name,VMAX,m,n,p,penalty,sp) :
Cette fonction est la fonction principale qui appelle toutes les autres. Elle calcule les matrices de transition et de coût à partir des informations nécessaires (fichier de données, dimensions du problème, constantes). L'utilisateur peut choisir de travailler au format 'sparse' ou non. La fonction renvoie également le temps d'exécution.

[Map_Data]=read_data(file_name,m,n) :
Cette fonction va chercher les informations dans le fichier de données et les formate en une matrice m $\times$n.

[Pos_Vector_Indexes,Speed_Vector_Indexes]=
index_computation() :
Cette fonction calcule les vecteurs d'index pour la position et la vitesse, comme vu plus haut. Elle utilise les variables globales VMAX et Map_Data.

[successor,cost]=compute_transitions(s,ax,ay) :
Cette fonction calcule le successeur d'un état et le coût de transition à partir d'un état d'origine et d'une accélération. Elle utilise les variables globales Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes, Finish_Data.

[s]=generate_starting_state() :
Cette fonction génère un état correspondant au quadruplet (x y 0 0) où le couple (x y) est une case de la ligne de départ. Elle utilise les variables globales Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes.

[Y_Vector, X_Vector, Probability_Vector, Cost_Vector]=
transition_matrix(ax,ay,p) :
Cette fonction calcule la matrice de transition correspondant à une accélération donnée, sous la forme de deux vecteurs de coordonnées, d'un vecteur de valeur et d'un vecteur de coût. Cette formulation a été retenue car, on l'a vu, P est très creuse et il vaut mieux la stocker au format 'sparse'. Elle utilise les variables globales Pos_Vector_Indexes et Speed_Vector_Indexes.

[result]=race_end(xt,yt,xt1,yt1) :
Cette fonction détermine si le segment [P1(xt,yt),P2(xt1,yt1)] coupe la ligne d'arrivée. Elle utilise les variables globales Map_Data, Pos_Vector_Indexes,

Speed_Vector_Indexes et Finish_Data.

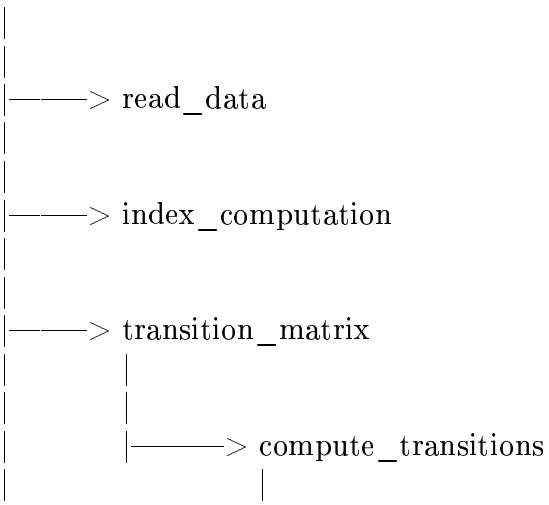
display_race(S) :
Cette fonction affiche le parcours correspondant à la série d'états S. Elle utilise les variables globales Map_Data et Policy.

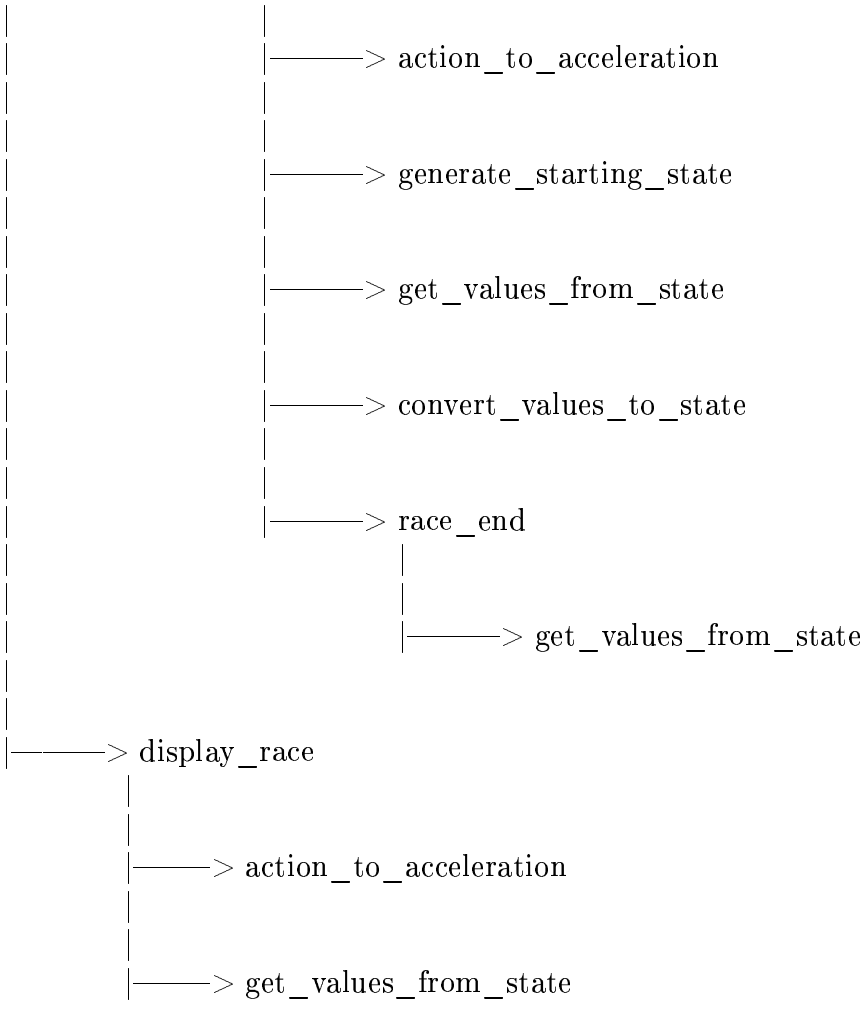
[ax,ay]=action_to_acceleration(a) :
Cette fonction transforme une 'action' (entier de 1:9) en l'accélération correspondante.

[s]=convert_values_to_state(x,y,vx,vy) :
Cette fonction transforme un quadruplet d'état en la variable d'état correspondante. Elle utilise les variables globales VMAX, Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes.

[x,y,vx,vy]=get_values_from_state(s) :
Cette fonction est l'inverse de la précédente: elle récupère le quadruplet à partir de la variable d'état. Elle utilise les variables globales VMAX, Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes.

3.3 Interaction des fonctions





3.4 Détail des fonctions

Nous allons maintenant aborder le détail des fonctions utilisées. Pour cela, nous remonterons l'arbre des dépendances vu ci-dessus en allant des fonctions les plus simples aux plus élaborées. Nous commencerons par les fonctions de passage quadruplet $\leftrightarrow$ singleton.

3.4.1 get_values_from_state

Cette fonction commence par déterminer les indices de position et de vitesse, autrement dit la place occupée par la vitesse/position dans l'ensemble ordonné des possibilités valides correspondantes. De par la manière dont est défini l'ordre, on a:

$$s = n\_speed * (position\_index - 1) + speed\_index$$

Nous arrivons donc à:

$$speed\_index = \text{(s modulo n\_speed)}$$

sauf si s=kn auquel cas on a : speed_index=n_speed. Une fois ces indices déterminés, nous allons pouvoir calculer les indices w_pos et w_speed, les places occupées par la vitesse/position dans l'ensemble ordonné de toutes les possibilités, grâce aux vecteurs d'indices. Une fois ceci fait, nous pouvons conclure puisque:

$$(x-1)m + y = w\_pos$$
$$(vx-1) * (2 * VMAX + 1) + vy = w\_speed$$

Il est à noter que les valeurs de la vitesse sont dans 1:(2VMAX+1). Il faut donc les recentrer sur -VMAX:VMAX.

3.4.2 convert_values_to_state

Cette fonction commence par déterminer les indices de position et de vitesse, autrement dit la place occupée par la vitesse/position dans l'ensemble ordonné des possibilités valides correspondantes. Pour ceci la fonction utilise les vecteurs d'indices, qui nous donnent les rangs i et j correspondant à x,y,vx,vy. On a:

$$Pos\_Vector\_Indexes(i) = (x-1) * m + y$$
$$Speed\_Vector\_Indexes(j) = (vx-1) * (2 * VMAX + 1) + vy$$

De la même manière, il est nécessaire de recentrer les valeurs des vitesses.

3.4.3 action_to_acceleration

Cette fonction fait une simple correspondance entre l'action et l'accélération, comme défini plus haut.

3.4.4 display_race

Cette fonction prend en argument un vecteur contenant une série d'états. Elle les transforme avec get_values_from_state et crée 4 vecteurs: X,Y,Ax,Ay qui correspondent aux coordonnées des états et aux accélérations correspondantes. A partir de X et Y on peut déterminer les transitions d'un état à l'autre. Grâce à la commande quiver de Matlab, on peut finalement afficher tous les vecteurs correspondant aux transitions et aux accélérations en surimpression sur la représentation du circuit.

3.4.5 race_end

Cette fonction détermine si une transition fait traverser la ligne d'arrivée. Elle commence par déterminer si l'état initial est suffisamment proche de la ligne d'arrivée pour lancer un calcul. Si il l'est, la fonction calcule alors l'intersection des deux droites formées par la ligne d'arrivée et la transition d'un état à l'autre. Si cette intersection appartient aux deux segments formés par la ligne d'arrivée et la transition, alors c'est que l'on a bien traversé la ligne d'arrivée.

3.4.6 generate_starting_state

Cette fonction très simple commence par sélectionner dans le vecteur d'indices de position les états correspondant à des cases de la ligne de départ, puis en tire une au hasard et finalement détermine la variable d'état correspondant à cette position et à une vitesse nulle.

3.4.7 compute_transitions

Cette fonction calcule le successeur d'un état, étant donné une action. Elle commence par vérifier les cas particuliers : la transition depuis l'état 'hors-piste' envoie directemment à une case aléatoire de la ligne de départ; la transition depuis l'état 'course terminée' renvoie directement 'course terminée' (état absorbant). Dans le cas normal, la fonction transforme la variable d'état en quadruplet, l'utilise en conjonction avec l'action pour calculer l'état successeur.

3.4.8 transition_matrix

Cette fonction détermine la matrice de transition associée à une seule action. Elle renvoie les résultats sous forme de 4 vecteurs, deux vecteurs donnant les coordonnées des éléments non nuls de la matrice, un vecteur donnant les valeurs de ces éléments non-nuls et le dernier donne les coûts associés à ces transitions. Ce format a été choisi car la matrice de transition, de par le modèle choisi, n'a au maximum que deux termes non nuls par ligne: la notation vectorielle, qui correspond au format 'sparse' (matrice creuse) de Matlab, s'imposait donc. La fonction calcule le successeur de chaque état sachant l'action choisie, le successeur sachant l'action 'problème de transition' et affecte les valeurs correspondantes (1-p et p). La fonction utilisée est compute_transitions. Le coût est déterminé en parallèle.

3.4.9 index_computation

Cette fonction calcule les vecteurs d'indices (cf. ci-dessus). Elle met la matrice de données sous forme d'un vecteur, et récupère les indices des éléments non nuls dans le vecteur d'indices. Elle fait la même chose pour le vecteur des indices de vitesse, à ceci près qu'elle doit d'abord calculer les combinaisons autorisées de Vx et Vy par rapport à la contrainte de vitesse maximale.

3.4.10 read_data

Cette fonction se contente d'aller lire dans un fichier la matrice décrivant le circuit et de la renvoyer.

3.4.11 gen_sp_transition_matrix

Cette fonction fait appel aux fonctions précédentes pour lire les données dans un fichier, faire le calcul des vecteurs d'indice et en déduire les matrices de transition pour chaque action. Ces matrices sont ensuite écrites sur le disque avec un format prédéterminé. Le fait que Matlab refuse de considérer des tableaux de matrices au format 'sparse' a déterminé ces choix de conception.

3.5 Problèmes et bugs de la version 1.1

La version 1.1 n'a pas de bugs connu, mais elle a tout de même un défaut rédhibitoire, à savoir que le codage n'a pas pris en compte le fait qu'il faut, avant de repasser la ligne de départ, avoir fait un tour complet. En effet, l'algorithme détermine que le plus court chemin vers la ligne d'arrivée est le demi-tour...

Chapitre 4

Version 1.2

4.1 Premières améliorations

4.1.1 Le problème du sens de parcours

On l'a vu, un des problèmes de la version 1.1 était le fait qu'aucun sens de parcours n'avait été défini, ce qui induisait l'algorithme de résolution en erreur. Par exemple, pour lui, le plus court chemin jusqu'à la ligne d'arrivée consiste à faire demi-tour dès le départ. Pour corriger ceci, on a défini un sens à la ligne d'arrivée: celle-ci devra être traversée de bas en haut et de gauche à droite. Dans le cas contraire, un traitement similaire à une sortie de piste est appliqué: une pénalité est subie et on doit recommencer depuis la ligne de départ. D'un point de vue pratique, on a défini un vecteur perpendiculaire à la ligne d'arrivée, dont le sens est donné par les contraintes bas-haut, gauche-droite. Lors d'une traversée de la ligne d'arrivée, on prend le produit scalaire du vecteur perpendiculaire à la ligne d'arrivée et du vecteur vitesse. Si il est positif, c'est que la traversée s'est faite dans le bon sens. Dans le cas contraire, la pénalité de demi-tour est appliquée.

4.1.2 Tests plus en profondeur et débuggage

Lors de l'implémentation du sens de course, il a fallu faire des tests. Par la même occasion, des tests approfondis sur l'ensemble du code ont été conduits, révélant plusieurs bugs, notamment dans les fonctions generate_starting_state et convert_values_to_state. Ces bugs ont bien entendu été corrigés.

4.1.3 Le problème du passage par le hors-piste

Dans la version 1.1, lors de la transition d'un état à un autre, le programme vérifiait la validité de la transition en vérifiant la validité de la case d'arrivée. Ceci ne prenait pas en compte la possibilité que, quelque part entre les cases de départ et d'arrivée, il était tout à fait possible qu'il y ait une case hors-piste. Pour pallier à ceci, lors des calculs de transitions, chaque case coupée par le segment passant par les deux points est vérifiée. Si une case hors-piste est trouvée, la transition est comptabilisée comme accident. Bien entendu, cette modification ralentit considérablement les calculs. Cependant, comme il suffit de calculer les matrices de coût et de transition une seule fois, cet inconvénient n'a pas été jugé rédhibitoire. La fonction calculant ceci se nomme safe_transition et est appelée par compute_transitions. Elle prend pour arguments (x,y) les coordonnées du point de départ, (x',y') les coordonnées du point d'arrivée et renvoie un booléen permettant de savoir si la transition est autorisée.

4.2 Modification de la MDP_Toolbox

4.2.1 Prise en compte des matrices creuses

La version 1.0 de la MDP_Toolbox n'est pas prévue pour l'utilisation du format 'sparse' (matrices creuses). Les tests précédents ont été conduits sur des matrices au format normal (full). Cependant, le problème automobile génère des matrices de transition et de coût de taille très importantes (plusieurs millions de termes, même pour un circuit de 10x10 cases). Les matrices en question sont très creuses (2 termes non nuls par ligne au maximum), d'où l'idée d'utiliser le format 'sparse' pour gérer les problèmes de taille. Pour pouvoir le faire, il va falloir modifier un peu le code de la MDP_Toolbox afin qu'elle puisse tourner avec des matrices au format 'sparse'. Matlab ne supportant pas les matrices 'sparse' à plus de 2 dimensions, il a fallu passer par une astuce: l'utilisation des tableaux de cellules. Il s'agit d'un type de tableau pouvant contenir des donnees de type quelconque, par exemple des matrices 'sparse'. Grâce à cela et moyennant quelques modifications, la toolbox MDP prend en compte les matrices au format 'sparse'.

Chapitre 5

Version 1.3: amélioration de la MDP Toolbox

5.1 Situation du problème

5.1.1 Pourquoi améliorer la Toolbox?

La MDP Toolbox est prévue pour résoudre des MDP quelconques. Cependant, dans le cadre de problèmes complexes, par exemple celui de la course automobile, la matrice de transitions (de taille S×S×A) peut devenir extrêmement grande et entraîner des problèmes de mémoire. Toujours dans l'exemple de course automobile, il est apparu que la matrice de transitions P était extrêmement creuse, d'où l'idée de modifier la Toolbox pour pouvoir utiliser des notations de matrices creuses (sparse). Ceci permet de traiter des problèmes bien plus grands dans le cas où P est assez creuse.

5.1.2 Le format 'matrice creuse' ou 'sparse'

Le format 'sparse' est un format spécial de matrices pour Matlab. Au lieu de stocker tous les coefficients de la matrice en séquence, on stocke 3 vecteurs X, Y et V correspondant aux coordonnées des éléments non nuls et à la valeur à stocker à ces coordonnées.

5.2 Implémentation

5.2.1 Introduction

Lors de la modification de la Toolbox, la première priorité a été que les changements soient le plus transparents possibles pour l'utilisateur. Les fonctions prennent indifféremment des arguments 'sparse' ou non. Dans le cas où les variables renvoyées sont des vecteurs, il n'y a pas de différence entre des arguments 'sparse' ou non. Dans le cas où la fonction renvoie une matrice, le type de la matrice correspond à celui des arguments. La seule fonction ayant subi un changement d'arguments est mdp_rand, où un argument a été rajouté pour laisser l'utilisateur choisir de générer un PDM 'sparse' ou non.

5.2.2 Problèmes d'implémentation

Les formats normal et 'sparse' sont interchangeables pour Matlab, il n'y aurait donc pas dû y avoir de difficultés. Cependant, le format 'sparse' ne permet pas de stocker des matrices en plus de 2 dimensions, ce qui est gênant dans le cas des PDM puisque P est en 3 dimensions. Il a donc fallu trouver une autre manière de faire. Ceci a été le seul problème majeur.

5.2.3 Résolution du problème

Le fait que les matrices 'sparse' ne puissent être en 3 dimensions a été contourné en utilisant les structures Matlab 'cell', qui sont des tableaux de dimensions quelconques pouvant contenir des objets quelconques. La syntaxe est légèrement plus complexe.

Pour accéder à l'élément P(i,j,k), on utilise la syntaxe P_sparse{k}(i,j).

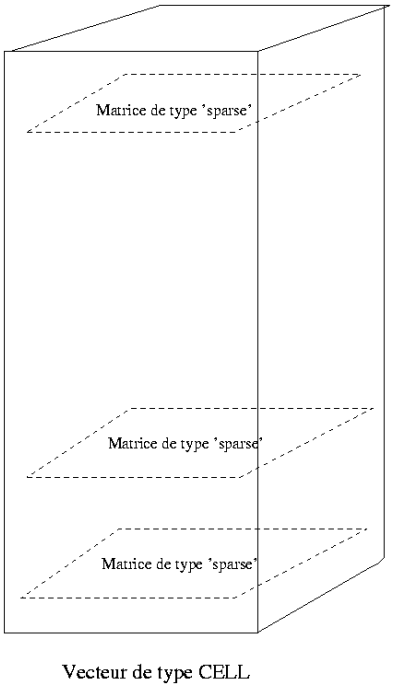


Figure 5.1: Le stockage 'sparse' en 3D.

5.2.4 Modification du code

Les modifications du code ont été relativement mineures, avec principalement l'ajout de tests sur le type de données traitées: normal ou 'sparse', lors des accès aux matrices. La fonction mdp_rand a été également légèrement modifiée avec un argument permettant de choisir le format des matrices fournies.

5.3 Exemple

Nous allons comparer les performances sur un exemple généré au hasard grâce à mdp_rand en forme normale, puis transformé en forme 'sparse'. Les deux versions de P et R passeront par la même fonction de résolution, puis nous comparerons les résultats.

```
% On génère le PDM
  [P,R]=mdp_rand(150,5,0);
% Résolution normale;
  [V, policy, iter, cpu_time] = mdp_value_iteration(P, R, 0.99);
% Transformation du problème en 'sparse'
  P2=cell(1,1,5);
  R2=cell(1,1,5);
  for t=1:5,
      P2{1,1,t}(:,:)=sparse(P(:,:,t));
      R2{1,1,t}(:,:)=sparse(R(:,:,t));
  end
% Résolution en 'sparse'
  [V2, policy2, iter2, cpu_time2] = mdp_value_iteration(P2, R2, 0.99);
```

Voiçi les résultats obtenus:

```
>> find(policy-policy2)
ans =
     []
>> max(V-V2)
ans =
   1.6653e-16
>> [cpu_time cpu_time2]
ans =
    0.1100    0.5500
```

5.3.1 Commentaire des résultats

On voit sur l'exemple précédent que le format de P et R n'influe pas sur la résolution puisque la politique et la fonction de valeur sont les mêmes dans les deux formats. Le format 'sparse' a mis cinq fois plus de temps à s'exécuter, ce qui s'explique par le fait que les PDM générés par mdp_rand sont très pleins. La formulation 'sparse' qui stocke les coordonnées en plus des valeurs fait de fait des appels plus nombreux. Cependant cette différence se gomme dans le cas de systèmes creux: voici par exemple les temps d'exécution dans

le cas de la course automobile, problème très creux comme on l'a vu.

```
>> [cpu_time cpu_time2]
ans =
    0.3700    0.3700
```

5.4 Conclusion

L'évolution de la MDP Toolbox permettant de prendre en compte les grands problèmes creux a été conçue pour être le plus transparente possible pour l'utilisateur. La seule contrainte imposée à celui-ci est le format des matrices 'sparse' en 3 dimensions lorsqu'il les utilise, c'est-à-dire un tableau de cellules de dimensions $1 \times 1 \times A$ dont les cellules contiennent les matrices 'sparse' correspondant à P(:,:,a), pour a=1:A.

Chapitre 6

Version 1.4: réécriture du code dans l'"esprit Matlab"

6.1 Introduction

Jusqu'ici les programmes de l'exemple de course automobile étaient écrits sans souci particulier d'utiliser des formulations vectorielles ou matricielles. Dans les cas où une telle possibilité était évidente, elle a été implémentée, mais il n'y a pas eu d'efforts particuliers pour tout vectoriser. Maintenant que le code tourne bien, la prochaine étape est justement la vectorisation des fonctions.

6.2 Présentation des changements apportés

Le passage d'une programmation impérative traditionnelle à une formulation matricielle est une évolution majeure du code. S'il est entendu qu'elle est conceptuellement plus compliquée, elle simplifie considérablement le code. Elle présente également l'avantage d'un temps de calcul bien plus faible, Matlab étant notoirement allergique aux boucles.
La structure du programme est restée la même: les fonctions gardent le même nom et le même effet. Bien entendu, les arguments et sorties ont été quelque peu modifiés, mais les modifications sont mineures.
Le plus grand changement a été la réécriture des fonctions en utilisant le calcul matriciel. Certaines fonctions s'y prêtaient bien, par exemple compute_transitions ou transition_matrix où les modifications ont été relative-

ment mineures. D'autres, en particulier race_end et safe_transition, ont plus posé problème, étant par nature confinées à une transition.

6.3 Récapitulatif des modifications

Voici une liste des fonctions utilisées, de leurs arguments et sorties, ainsi qu'une brève description des modifications apportées.

Les variables globales utilisées sont:
VMAX la vitesse maximale autorisée;
Map_Data la matrice correspondant au circuit;
Pos_Vector_Indexes le vecteur des indices de position;
Speed_Vector_Indexes le vecteur des indices de vitesse;
Finish_Data le vecteur contenant les informations sur la ligne d'arrivée et
Policy la politique déterminée par la résolution du PDM.

[P,C,cpu_time]=
gen_transition_matrix(file_name,VMAX,m,n,p,penalty,sp) :
Pas de différence avec son incarnation précédente, en effet elle traite déjà avec des vecteurs et matrices..

[Map_Data]=read_data(file_name,m,n) :
Identique à la première version.

[Pos_Vector_Indexes,Speed_Vector_Indexes]=
index_computation() :
Identique à la première version.

[successor,cost]=compute_transitions(s,ax,ay) :
Cette fonction présente de nombreuses différences par rapport à sa première incarnation; en effet, elle passe du calcul d'un seul successeur au calcul d'un vecteur de successeurs. Le procédé utilisé consiste à faire des tests de validité successifs pour chaque possibilité d'action interdite sur tout le vecteur, puis de filtrer les résultats pertinents à l'aide de masques. Elle utilise les variables globales Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes, Finish_Data et VMAX.

[s]=generate_starting_state() :
Identique à la première version.

[Y_Vector, X_Vector, Probability_Vector, Cost_Vector]= transition_matrix(ax,ay,p) :
Cette fonction a subi assez peu de changements, principalement des astuces de calcul matriciel pour mettre en forme les sorties. Elle utilise les variables globales Pos_Vector_Indexes et Speed_Vector_Indexes.

[result]=race_end(xt,yt,xt1,yt1) :
Cette fonction a subi une refonte similaire à compute_transitions. Elle fait également usage de masques permettant de traiter les informations pertinentes du vecteur. Elle utilise les variables globales Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes et Finish_Data.

display_race(S) :
Quasi-identique à la première version.

[ax,ay]=action_to_acceleration(a) :
Identique à la première version.

[s]=convert_values_to_state(x,y,vx,vy) :
Quasi-identique à la première version: l'algorithme prend des vecteurs sans problème. Elle utilise les variables globales VMAX, Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes.

[x,y,vx,vy]=get_values_from_state(s) :
Quasi-identique à la première version: l'algorithme prend des vecteurs sans problème. Elle utilise les variables globales VMAX, Map_Data, Pos_Vector_Indexes, Speed_Vector_Indexes.

[res]=safe_transition(xx,yy,x,y) :
Cette fonction vérifie que les transitions proposées ne passent pas par des états hors-piste. Pour cela elle calcule la droite associée à chaque couple de points début/fin de transition, puis vérifie chaque point sur cette droite. Il suffit d'un point hors-piste pour que la transition correspondante échoue. Elle utilise les variables globales VMAX et Map_Data.

6.4 Performances comparées avec la version précédente

La réécriture du code en versions 1.3 puis 1.4 a permis des améliorations de performance spectaculaires, notamment sur les grands problèmes; par exemple, le tableau suivant donne les temps d'exécution, à paramètres identiques, du calcul de P et C pour les versions 1.1, 1.3 et 1.4. Il est à noter que la version 1.3 utilise le format 'sparse' pour les calculs, ce qui les accélère déjà considérablement.

Taille du circuit:	4x4	7x7	10x10	25x25	50x50	100x100
Version 1.1 (s):	3.76	11.34	18.89	OutOfMem	OutOfMem	OutOfMem
Version 1.3 (s):	0.39	0.91	1.39	10.49	48.28	>1000
Version 1.4 (s):	0.38	0.90	1.45	9.10	33.76	246.13

Le tableau ci-dessus montre clairement que le passage des matrices normales en matrices creuses (de 1.1 à 1.3) a été une évolution majeure puisque les temps d'exécution ont été divisés par un facteur supérieur à 10 pour les petits problèmes et permet également le traitement de problèmes de grande taille. Le passage de la version 1.3 à la version 1.4 est essentiellement une optimisation; les bénéfices du nombre d'appels de fonctions réduits deviennent visibles pour les problèmes de grande taille, où les temps d'exécution sont sérieusement réduits.

6.5 Possibilités d'évolution vers une nouvelle version

On l'a vu, la version 1.4 a partiellement optimisé le temps de calcul pour les grands problèmes. Il serait cependant possible de l'améliorer encore. La fonction la plus gourmande en temps de calcul est convert_values_to_state, à cause d'une boucle for qui pourrait peut-être être éliminée. De plus, lors du calcul des transitions, les fonctions calculant les transitions interdites (par exemple safe_transition) sont appliquées au vecteur d'état entier. Il serait peut-être possible d'optimiser cela en n'envoyant que les données pertinentes, ce qui poserait des problèmes d'index.
En l'état actuel des choses, une version 1.5 n'est pas envisagée, car les modifications envisagées ne seraient effectuées que dans le but d'optimiser le temps

d'exécution qui est déjà satisfaisant. La suite du stage se concentrera plutôt sur l'implémentation d'algorithmes de résolution avancés, le TD(λ) et la Q-learning.

Chapitre 7

L'algorithme TD(λ)

7.1 Aspect théorique

Soit $\lambda \in [0,1]$ un paramètre de pondération. L'algorithme de TD(λ) défini par Sutton est le suivant :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=N-1} \lambda^{m-k} d_m, \quad k = 0, \ldots, N-1 \tag{7.1}$$

On peut essayer de mieux comprendre le rôle du coefficient λ en reécrivant l'équation 7.1 sous la forme

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(z_k^\lambda - V(s_k))$$

On a alors

$$\begin{aligned} z_k^\lambda &= V(s_k) + \sum_{m=k}^{m=N-1} \lambda^{m-k} d_m \\ &= V(s_k) + d_k + \lambda \sum_{m=k+1}^{m=N-1} \lambda^{m-k-1} d_m \\ &= V(s_k) + d_k + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= V(s_k) + r_k + V(s_{k+1}) - V(s_k) + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= r_k + (\lambda z_{k+1}^\lambda + (1-\lambda)V(s_{k+1})) \end{aligned}$$

Ainsi, le cas $\lambda = 1$ revient à prendre en compte le coût total observé de la trajectoire, comme dans la méthode de Monte Carlo, et le cas $\lambda = 0$ revient

à ne considérer qu'un horizon de un coup, comme pour la méthode de la programmation dynamique.

Pour tout λ, les deux approches de type *first-vist* ou *every-visit* peuvent être considérée. De même, une version *on-line* de l'algorithme d'apprentissage TD(λ) décrit par l'équation 7.1 est possible :

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\lambda^{k-l}d_k, \quad l = 0, \ldots, k \tag{7.2}$$

dès que la transition (s_k, s_{k+1}, r_k) est simulée et l'erreur d_k calculée.

La convergence presque sûre de l'algorithme TD(λ) a été montrée pour toute valeur de λ, en *on-line* ou *off-line*, sous les hypothèses classiques de visite en nombre infini de chaque état $s \in S$, et décroissance des α vers 0 à chaque itération n, telle que $\sum_n \alpha_n(s) = \infty$ et $\sum_n \alpha_n^2(s) < \infty$ [**?**, **?**].

Il est à noter que l'effet du λ est encore mal compris, et sa détermination optimale pour un problème donné reste très empirique : une valeur intermédiaire entre 0 et 1, proche de 0,7, est souvent employée...

L'application du TD(λ) pour l'évaluation d'une politique π selon le critère γ-pondéré entraine certaines modifications des algorithmes standarts 7.1 ou 7.2, qu'il est nécessaire de citer ici.

Un calcul en tout point semblable au cas $\gamma = 1$ conduit à une règle du type :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=\infty} (\gamma\lambda)^{m-k} d_m \tag{7.3}$$

Il est alors clair que l'absence potentielle d'états finaux absorbants rend inadéquate un algorithme de type *off-line* ne mettant à jour la fonction de valeur V qu'à la fin de la trajectoire, car celle ci peut être de taille infinie. On définit donc une version *on-line* de 7.3, qui prend la forme suivante :

$$V(s) \leftarrow V(s) + \alpha(s)z_n(s)d_n, \quad \forall s \in S, \tag{7.4}$$

dès que la $n$ième transition (s_n, s_{n+1}, r_n) a été simulée et l'erreur d_n calculée. Le terme $z_n(s)$, dénommé trace d'éligibilité [1] se définit ainsi dans la version la plus proche de l'algorithme TD(λ) original :

[1] Traduit de l'anglais *eligibility trace*, ou encore *activity*.

Trace d'éligibilité accumulative

$$\begin{aligned} z_0(s) &= 0, \quad \forall s \in S \\ z_n(s) &= \begin{cases} \gamma\lambda z_{n-1}(s) & \text{si } s \neq s_n \\ \gamma\lambda z_{n-1}(s) + 1 & \text{si } s = s_n \end{cases} \end{aligned}$$

Ce coefficient d'éligibilité augmente donc sa valeur à chaque nouveau passage dans l'état associé, puis décroit exponentiellement au cours des itérations suivantes, jusqu'à un nouveau passage dans cet état (voir figure 7.1).

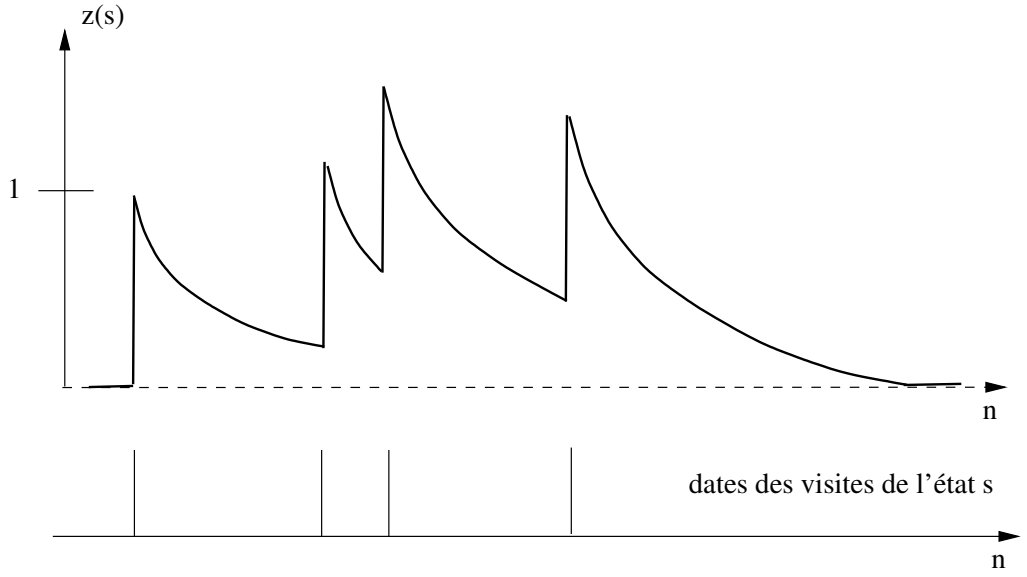


Figure 7.1: Trace d'éligibilité cumulative.

Dans certains cas une définition légèrement différente de la trace $z_n(s)$ semble conduire à une convergence plus rapide de la fonction de valeur V :
Trace d'éligibilité avec réinitialisation

$$\begin{aligned} z_0(s) &= 0, \quad \forall s \in S \\ z_n(s) &= \begin{cases} \gamma\lambda z_{n-1}(s) & \text{si } s \neq s_n \\ 1 & \text{si } s = s_n \end{cases} \end{aligned}$$

La valeur de la trace est donc saturée à 1, comme le montre la figure 7.2.

Une implémentation directe de TD(λ) basée sur la trace d'éligibilité n'est bien sûr pas efficace dès que la taille de l'espace d'état S devient trop grande. Une première solution approchée consiste à forcer à 0 la valeur de toutes les

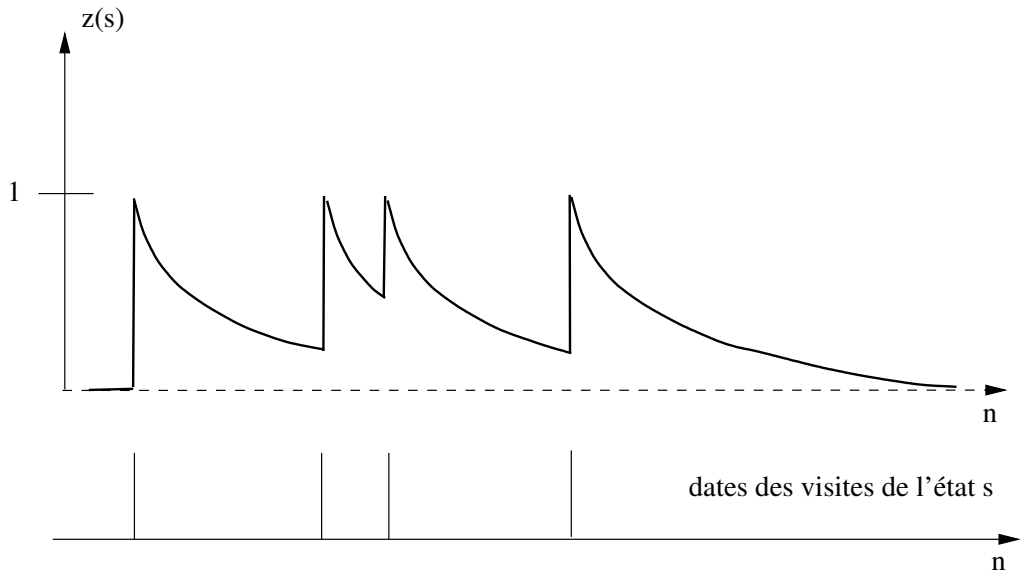


Figure 7.2: .Trace d'éligibilité avec réinitialisation

traces $z_n(s) < \varepsilon$, et donc à ne maintenir que les traces des états récemment visités (plus précisément, on cesse de maintenir un état dont la dernière visite remonte à plus de $\frac{\log(\varepsilon)}{\log(\gamma\lambda)}$) transitions.

Une autre méthode approchée connue sous le nom de *truncated temporal differences*, revient à gérer un horizon glissant de taille m mémorisant les derniers états visités et à mettre à jour sur cette base à chaque itération n la valeur de l'état visité à l'itération $(n - m)$.

7.2 Modélisation

Dans un premier temps, l'algorithme TD(λ) a été prévu pour être appliqué à l'exemple de course de voitures. Par conséquent il n'était pas spécialement générique puisque les trajectoires suivies étaient finies et de structure semblable.
Dans un deuxième temps, il a fallu raffiner et généraliser l'algorithme en vue de son intégration dans la MDP Toolbox.
Une fois ceci réalisé, une série de tests de validation ont été effectués.

7.3 Implémentation

7.3.1 Implémentation liée à la course automobile

De manière similaire à ce qui a déjà été fait, l'implémentation s'est faite en deux temps, une implémentation "naïve" destinée à se familiariser avec l'algorithme puis une implémentation matricielle mettant à profit les particularités de Matlab.

7.3.2 Implémentation générique

L'évolution en implémentation générique s'est faite de manière relativement aisée, la seule difficulté majeure étant la nécessité de pouvoir calculer des trajectoires pour des PDM complètement différents, sans toucher au code de la fonction. Ceci a été réalisé grâce à la fonction "feval" de Matlab: le nom d'une fonction calculant les trajectoires est donné en argument, et cette fonction est à son tour appelée grâce à feval. La seule contrainte imposée à l'utilisateur est un format d'appel fixé, c'est donc à lui de fournir une fonction de trajectoire reconnaissable par la fonction TD(λ).

7.4 Résultats obtenus

L'utilisation de la fonction TD(λ) n'est pour l'instant pas entièrement satisfaisante. En effet, l'évaluation de V, bien que proche de la valeur réelle dans certains cas, n'a rien à voir dans d'autres. De plus, dans tou les cas, l'erreur ne semble pas diminuer avec le nombre de simulations. Il reste donc du travail à faire avant de pouvoir l'intégrer à la MDP Toolbox.

Chapitre 8

L'algorithme Q-learning

8.1 Aspect théorique

Les méthodes indirectes permettent de résoudre des PDMs dont on ne connaît pas le modèle, à la condition de pouvoir expérimenter ou simuler ce modèle. Ces méthodes présentent toutefois un inconvénient : elles nécessitent de stocker au moins partiellement les fonctions $\hat{p}$ et $\hat{r}$, ce qui peut nécessiter jusqu'à $O(|S|^2|A|)$ d'espace de stockage.

L'algorithme Q-learning permet de se passer du stockage de $\hat{p}$ et $\hat{r}$, remplacé par le stockage de $\hat{Q}$, de taille $O(|S||A|)$. La contrepartie de ce gain en stockage est que des expérimentations / simulations plus nombreuses sont nécessaires : puisqu'on n'a pas de modèle, on ne peut effectuer les k mises à jour supplémentaires des algorithmes de la famille Dyna. Le choix d'une méthode directe sera donc préféré lorsque les simulations ont un coût faible, et qu'un problème de taille mémoire peut se poser.

Le principe de l'algorithme Q-learning est d'approcher de manière itérative la solution Q du système d'équations suivant :

$$Q(s,a) = \sum_{s' \in S} p(s'|s,a) \cdot \{r(s,a,s') + \gamma \cdot \max_{a' \in A} Q(s',a')\}. \tag{8.1}$$

L'itération de base de Q-learning est la suivante :

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha(r + \gamma \max_{a' \in A} \hat{Q}(s',a') - \hat{Q}(s,a)), \tag{8.2}$$

itération effectuée chaque fois qu'une transition $< s,a,s',r >$ est observée (par simulation ou expérimentation). Le coefficient α est un "taux d'apprentissage"

qui décroît vers 0 avec le nombre d'expérimentations réalisées : plus le nombre d'itérations effectuées augmente, plus on fait confiance à l'estimation courante de Q, et moins les expériences à venir peuvent la modifier.

Sous certaines conditions peu exigeantes, on a montré la convergence de l'algorithme Q-learning vers la vraie valeur de Q (encore une fois, on doit s'assurer que les expériences / simulations permettent de visiter tous les couples états / actions un grand nombre de fois).

8.2 Modélisation

Dans un premier temps, l'algorithme Q-learning a été prévu pour être appliqué à l'exemple de course de voitures. Par conséquent il n'était pas spécialement générique puisque les trajectoires suivies étaient finies et de structure semblable.
Dans un deuxième temps, il a fallu raffiner et généraliser l'algorithme en vue de son intégration dans la MDP Toolbox.
Des tests de validation ont également été réalisés sur cette fonction.

8.3 Implémentation

8.3.1 Implémentation liée à la course automobile

De manière similaire à ce qui a déjà été fait, l'implémentation s'est faite en deux temps, une implémentation "naïve" destinée à se familiariser avec l'algorithme puis une implémentation matricielle mettant à profit les particularités de Matlab.

8.3.2 Implémentation générique

L'évolution en implémentation générique s'est aussi faite de manière relativement aisée, la seule difficulté majeure étant la nécessité de pouvoir calculer des transitions pour des PDM complètement différents, sans toucher au code de la fonction. Ceci a été réalisé grâce à la fonction "feval" de Matlab: le nom d'une fonction calculant les transitions est donné en argument, et cette fonction est à son tour appelée grâce à feval. La seule contrainte imposée à l'utilisateur est un format d'appel fixé, c'est donc à lui de fournir une fonction

de transition reconnaissable par la fonction Q-learning.

8.4 Résultats obtenus

L'utilisation de la fonction Q-learning n'est pour l'instant pas non plus entièrement satisfaisante. L'algorithme a l'air de converger, mais nécessite énormément de simulations avant d'arriver à une politique proche de la politique optimale. Le calcul de V laisse également à désirer, mais il ne s'agit probablement que d'un petit bug. Il reste donc aussi du travail à faire avant de pouvoir l'intégrer à la MDP Toolbox.

8.5 Conclusion

Lors de ce stage j'ai pu participer à l'élaboration de fonctions Matlab destinées à être incorporées à une boîte à outils, approfondir ma connaissance du domaine des chaînes en découvrant les PDM et les algorithmes associés. Chose non négligeable, ma maîtrise de Matlab a aussi grandement progressé. Je garderai un excellent souvenir de ce stage, qui fut une expérience très positive.

Chapitre 9

Bibliographie

Régis SABBADIN: Test traité IC3 - Chapitre PDM. Une partie de l'introduction et le chapitre sur la théorie des PDM ont été tirés de cet ouvrage.

Frédérick GARCIA: Traité HERMES Apprentissage Machine. La partie théorique de l'algorithme TD(λ) a été tirée de cet ouvrage.

J.M.ALLIOT, T.SCHIEX, P.BRISSET, F.GARCIA: Intelligence Artificielle et Informatique Théorique (2ème édition), Editions Cépaduès.